CSC4001 Softeware Engineering Project Report

Name: Xiang Fei

Student ID: 120090414

1. Differential Testing

1a. Overview

In this task, we need to find bugs in a buggy PIG-interpreter through differential testing. More specifically, a generator for the PIG language and a correct version of PIG-interpreter are required to be implemented. From my perspective, the core idea of differential testing is to execute two functional-equivalent programs with the same input. For our task, the two programs is the buggy PIG-interpreter and our implemented interpreter or the standard interpreter, the input is the PIG code generated by our implemented generator.

1b. Code Implementation of the PIG Code Generator

> My implementation only import random library.

(1) Function form_basic_expression(existing_vars)

This function is used to randomly generate different kinds of basic expressions. Here, "basic" means the element in the expression is either variables or constants. For example:

- 00000011 is a basic expression since it is a constant.
- ! (v000) is a basic expression of NOT operation since the element v000 is a variable.
- (00000000) & (v000) is a basic expression of AND operation since the element 00000000 is a constant, and the element v000 is a variable.
- (v000) | (v001) is a basic expression of OR operation since the elements v000 and v001 are both variables.
- (00000011) + (v001) is a basic expression of ADD operation since the element 00000011 is a constant, and the element v001 is a variable.
- (00000011) - (v001) is a basic expression of SUB operation since the element 00000011 is a constant, and the element v001 is a variable.

The input of the function is a list of existing variables.

For "CONSTANT" case, just randomly generate each bit with 0 or 1 under the variable type.

```
if operator == "CONSTANT":
        var_type = random.choice(var_types)
        bits = int(var_type[2:])
        exp = random.choice(["0", "1"]) * bits
```

For "NOT" case, randomly choose the element to be constant or a variable, if constant, randomly generate it, if variable, randomly choose it from the existing variables.

```
if operator == "NOT":
        flag = random.choice([0,1])
        if flag == 1: # use constant
            var_type = random.choice(var_types)
            bits = int(var_type[2:])
            val = random.choice(["0", "1"]) * bits
            exp = f"! ( {val} )"
        else: # use var
            var_name = random.choice(existing_vars)
            exp = f"! ( {var_name} )"
```

For "AND" case, randomly choose the left element and right element to be constant or variable, if constant, randomly generate it, if variable, randomly choose it from the existing variables.

```
if operator == "AND":
        flag_left = random.choice([0,1])
        flag_right = random.choice([0,1])
        if flag_left == 1:
            var_type = random.choice(var_types)
            bits = int(var_type[2:])
            val = random.choice(["0", "1"]) * bits
            exp_left = val
        else:
            var_name = random.choice(existing_vars)
            exp_left = var_name
        if flag_right == 1:
            var_type = random.choice(var_types)
            bits = int(var_type[2:])
            val = random.choice(["0", "1"]) * bits
            exp_right = val
        else:
            var_name = random.choice(existing_vars)
            exp_right = var_name
        exp = f"( {exp_left} ) & ( {exp_right} )"
```

For "OR" case, randomly choose the left element and right element to be constant or variable, if constant, randomly generate it, if variable, randomly choose it from the existing variables.

```
if operator == "OR":
        flag_left = random.choice([0,1])
        flag_right = random.choice([0,1])
        if flag_left == 1:
            var_type = random.choice(var_types)
            bits = int(var_type[2:])
            val = random.choice(["0", "1"]) * bits
```

```
            exp_left = val
        else:
            var_name = random.choice(existing_vars)
            exp_left = var_name
        if flag_right == 1:
            var_type = random.choice(var_types)
            bits = int(var_type[2:])
            val = random.choice(["0", "1"]) * bits
            exp_right = val
        else:
            var_name = random.choice(existing_vars)
            exp_right = var_name
        exp = f"( {exp_left} ) | ( {exp_right} )"
```

For "ADD" case, randomly choose the left element and right element to be constant or variable, if constant, randomly generate it, if variable, randomly choose it from the existing variables.

```
if operator == "ADD":
        flag_left = random.choice([0,1])
        flag_right = random.choice([0,1])
        if flag_left == 1:
            var_type = random.choice(var_types)
            bits = int(var_type[2:])
            val = random.choice(["0", "1"]) * bits
            exp_left = val
        else:
            var_name = random.choice(existing_vars)
            exp_left = var_name
        if flag_right == 1:
            var_type = random.choice(var_types)
            bits = int(var_type[2:])
            val = random.choice(["0", "1"]) * bits
            exp_right = val
        else:
            var_name = random.choice(existing_vars)
            exp_right = var_name
        exp = f"( {exp_left} ) + ( {exp_right} )"
```

For "SUB" case, randomly choose the left element and right element to be constant or variable, if constant, randomly generate it, if variable, randomly choose it from the existing variables.

```
if operator == "SUB":
        flag_left = random.choice([0,1])
        flag_right = random.choice([0,1])
        if flag_left == 1:
            var_type = random.choice(var_types)
            bits = int(var_type[2:])
            val = random.choice(["0", "1"]) * bits
            exp_left = val
```

```
        else:
            var_name = random.choice(existing_vars)
            exp_left = var_name
        if flag_right == 1:
            var_type = random.choice(var_types)
            bits = int(var_type[2:])
            val = random.choice(["0", "1"]) * bits
            exp_right = val
        else:
            var_name = random.choice(existing_vars)
            exp_right = var_name
        exp = f"( {exp_left} ) - ( {exp_right} )"
```

(2) Function form_complex_expression(existing_vars)

This function is used to randomly generate different kinds of complex expressions. Here, "complex" means the element in the expression is a basic expression. So, just use form_basic_expression function to generate each element in the function.

```
def form_complex_expression(existing_vars):
    operators = ["NOT", "AND", "OR", "ADD", "SUB"]
    operator = random.choice(operators)
    if operator == "NOT":
        exp1 = form_basic_expression(existing_vars)
        exp = f"! ( {exp1} )"
    if operator == "AND":
        exp1 = form_basic_expression(existing_vars)
        exp2 = form_basic_expression(existing_vars)
        exp = f"( {exp1} ) & ( {exp2} )"
    if operator == "OR":
        exp1 = form_basic_expression(existing_vars)
        exp2 = form_basic_expression(existing_vars)
        exp = f"( {exp1} ) | ( {exp2} )"
    if operator == "ADD":
        exp1 = form_basic_expression(existing_vars)
        exp2 = form_basic_expression(existing_vars)
        exp = f"( {exp1} ) + ( {exp2} )"
    if operator == "SUB":
        exp1 = form_basic_expression(existing_vars)
        exp2 = form_basic_expression(existing_vars)
        exp = f"( {exp1} ) - ( {exp2} )"
    return exp
```

(3) Function related with branch structure

In my implementation, I considered both forward jump and backward jump.

- For the forward jump, the structure is like the following, just jump to the 5 lines forward with a fixed setting.

```
B 17 ( 00000001 )
A v002 ( 00000001 )
A v002 ( 00000001 )
A v002 ( 00000001 )
A v002 ( 00000001 )
A v002 ( 00000001 )
O v002
O v002
```

The code is also following a direct way.

```
# forward jump , fixed settings, 5 lines forward
forward_branch_flag = random.randint(1,5)
if forward_branch_flag == 1 and (line_nums+8) < 950:
    stmt = f"B {line_nums + 6} ( 00000001 )"
    print(stmt, file=f)
    line_nums += 1
    for i in range(5):
        stmt = f"A {var_name} ( 00000001 )"
        print(stmt, file=f)
        line_nums += 1
    stmt = f"O {var_name}"
    print(stmt, file=f)
    line_nums += 1
    stmt = f"O {var_name}"
    print(stmt, file=f)
    line_nums += 1
```

- For the backward jump, the structure is like the following, also follows a fixed setting.

```
D bv8 v008
D bv8 v009
A v009 ( ( v009 ) + ( 00000001 ) )
A v008 ( ( v004 ) & ( 00000000000000000000000000000000 ) )
B 007 ( ( v009 ) - ( 00001001 ) )
O v008
```

The expression of the assign instruction in the backward branch structure is a basic expression. The code is like the following.

```
def branch_structure(var_name1, var_name2, existing_vars):
    stmt = f"A {var_name2} ( ( {var_name2} ) + ( 00000001 ) )"
    print(stmt, file=f)
    stmt = f"A {var_name1} ( {form_basic_expression(existing_vars)} )"
    print(stmt, file=f)
```

```
branch_flag = random.randint(1,5)
if branch_flag == 1 and line_nums < 950:
    var_type = "bv8"
    var_name1 = "v" + str(var_nums+1).zfill(3)
    var_name2 = "v" + str(var_nums+2).zfill(3)
    var_nums += 2
    existing_vars.append(var_name1)
    existing_vars.append(var_name2)
    stmt = f"D {var_type} {var_name1}"
    print(stmt, file=f)
    line_nums += 1
    stmt = f"D {var_type} {var_name2}"
    print(stmt, file=f)
    line_nums += 1
    branch_line = str(line_nums).zfill(3)
    branch_structure(var_name1, var_name2, existing_vars)
    line_nums += 2
    stmt = f"B {branch_line} ( ( {var_name2} ) - ( {eight_bit_constant()} ) )"
    print(stmt, file=f)
    line_nums += 1
    stmt = f"O {var_name1}"
    print(stmt, file=f)
    line_nums += 1
```

During the main program flow, the generation of declaration, assignment, output, branch, remove are all random.

1c. Code Implementation of the PIG-Interpreter

My implementation of the interpreter is designed to fit my design of the code generator. I implemented several functions to compute the value of basic expressions. For complex expression, we can first compute its element using the functions that obtain the value of basic expressions, and then the complex expression can be treated as a basic expression.

Function to compute basic not, the input vars is a dict to store the variable type and value, expression is the inner expression, either constant or variable:

```
def compute_basic_not(vars,expression): # NOT-VAR or NOT-CON
    if expression[0] == "v": # NOT-VAR
        exp_value = vars[expression][1]
        not_exp_value = ''.join('1' if bit == '0' else '0' for bit in exp_value)
        return not_exp_value
    else: # NOT-CON
        not_exp_value = ''.join('1' if bit == '0' else '0' for bit in expression)
        return not_exp_value
```

Function to compute basic and, Function to compute basic not, the input vars is a dict to store the variable type and value, left_exp and right_exp is the element in the left and right side respectively, either constant or variable. I also handled the bit length differences cases:

```
def compute_basic_and(vars,left_exp,right_exp):
    if left_exp[0] == "v":
        left_type = vars[left_exp][0]
        left_bits = int(left_type[2:])
        left_exp_value = vars[left_exp][1]
    else:
        left_bits = len(left_exp)
        left_exp_value = left_exp
    if right_exp[0] == "v":
        right_type = vars[right_exp][0]
        right_bits = int(right_type[2:])
        right_exp_value = vars[right_exp][1]
    else:
        right_bits = len(right_exp)
        right_exp_value = right_exp
    if left_bits == right_bits:
        result = bin(int(left_exp_value, 2) & int(right_exp_value, 2))
[2:].zfill(left_bits)
        return result
    elif left_bits > right_bits:
        right_exp_value = right_exp_value.zfill(left_bits)
        result = bin(int(left_exp_value, 2) & int(right_exp_value, 2))
[2:].zfill(left_bits)
        return result
    else:
        left_exp_value = left_exp_value.zfill(right_bits)
        result = bin(int(left_exp_value, 2) & int(right_exp_value, 2))
[2:].zfill(right_bits)
        return result
```

Functions to compute basic or, add, and sub are similar as the above one.

Besides, when using assignment, the assigned variable has its own type, so the following function stored_value is used to transform the results of expression computation to the stored result of the assigned variable:

```
def stored_value(bits,expression):
    exp_len = len(expression)
    if exp_len < bits:
        decimal_num = int(expression,2)
        exp_value = bin(decimal_num)[2:].zfill(bits)
        return exp_value
    elif exp_len > bits:
        expression = expression[-bits:]
        exp_value = expression
        return exp_value
    else:
        exp_value = expression
        return exp_value
```

2. Metamorphic Testing

2a. Overview

In this task, we need to find bugs in buggy PIG-interpreters through metamorphic testing. More specifically, a generator for the PIG language and checker are required to be implemented. From my perspective, the core idea of metamorphic testing is modify the input file, and send different files with some relations to the buggy interpreter, then compare the output files to find whether there might be some problems with the testing interpreter.

2b. Code Implementation of gen_meta.py

Actually, the implementation is very similar to the differential testing, also follows the basic and complex expression pattern. My design for the metamorphic testing is to generate different input files which should obtain the same output result. Therefore, in input2, I did the following modification compared with input1:

- change the expression in input1 with two not operation.
- switch the left expression and right expression in the AND, OR, ADD operations in input1.
- in input1, I may assign 0 to some declared variables, but in input2, I just use declaration without direct zero assignment, this can check whether the interpreter can directly assign 0 to the just declared variables.

2c. Code Implementation of checker.py

Under my design, the checker.py script just needs to determine whether the output file 1 and 2 are the same, if so, write 0 to the res.out file, otherwise write 1.

3. Dataflow Analysis

3a. Overview

In this task, we need to find the number of lines that contain undeclared variables. Besides, we need to treat each branch statement as it may both branch or not branch to the target line number every time it is executed no matter what the expression is.

3b Code Implementation of Dataflow Analysis

In my implementation, there is a important variable named vars, which is a dict to store whether a PIG-variable is declared or not, the key is the var_name, the value is a boolean value, 1 means declared and 0 means undeclared. Therefore, I implement a check_declare function as follows:

```
def check_declare(vars, var_name):
    try:
        if vars[var_name] == 1:
            return True
        else:
            return False
    except:
        return False
```

Also, I use a list undeclared_lines to record the lines with undeclared variables, and finally can use it to compute the number of undeclared lines. Then, in the main program, I deal with different cases with different tokens, For "D", just assign the dict value of the corresponding variable to be 1, as follows:

```
if token == "D":
    vars[line[-4:]] = 1
```

For "A", just check whether the used variables is declared or not by searching "v" character and using the check_declare function.

```
elif token == "A":
    index = line.find("v")
    var_name = line[index:index+4]
    if check_declare(vars, var_name) != True:
        undeclared_lines.append(i)
    else:
        while index != -1:
            index = line.find("v", index+1)
            if index == -1:
                break
            var_name = line[index:index+4]
            if check_declare(vars, var_name) != True:
                undeclared_lines.append(i)
                break
```

For "R", just modified the dict value of the corresponding variable to be 0, as follows.

```
elif token == "R":
    var_name = line[-4:]
    if check_declare(vars, var_name) != True:
        undeclared_lines.append(i)
    if var_name in vars:
        vars[var_name] = 0
```

For "O", just check whether the variable is declared or not.

For "B", this actually is the most difficult part of the work. I first check whether the variables used in the expression are declared or not, and then, we need to deal with the branch logic.

We need to pay attention that, if we just treat every branch instruction as jump or not, then there will be too many possibilities and the time complexity is actually not reasonable and for some test cases, the program execution can not be finished within 10 seconds. Therefore, my strategy is:

- For all the forward branches, consider both branch or not branch.
- For the backward branches, in the iteration of a former branch, the later branch will not be considered, since the case from the line of former branch to the line of later branch can be considered in the main

program. In the iteration of later branch, only if the target of the former branch is smaller then the target of the later branch, the former branch will be considered. This is because from the line of the later branch to the target of the former branch will not be considered in the main program, so we need to deal with these cases.

My implementation follows a recursive way. There are two recursive functions **forward_branch_execute** and **backward_branch_execute**, and for each iteration, we have its own vars dict variable *new_var*, which is a deep copy of the previous iteration, to record the variable declaration situation in this current iteration. The deep copy is implemented as follows:

```
def deep_copy_dict(d):
    new_dict = {}
    for key, value in d.items():
        if isinstance(value, dict):
            new_dict[key] = deep_copy_dict(value)
        else:
            new_dict[key] = value
    return new_dict
```

And in both **forward_branch_execute** and **backward_branch_execute**, still process the "D", "A", "R", "O" with *new_var*, and for "B", use my strategy and follows a recursive way, like the following:

```
# in forward_branch_execute function
inner_target_line = int(line_j[2:5])
new_vars = deep_copy_dict(vars)
if inner_target_line > j:
    if len(back_lines) != 0:
        undeclared_lines =
forward_branch_execute(new_vars,undeclared_lines,inner_target_line,current_line,ba
ck_lines)
    else:
        undeclared_lines =
forward_branch_execute(new_vars,undeclared_lines,inner_target_line,j,back_lines)
elif inner_target_line < j:
    if len(back_lines) != 0:
        if j < current_line and inner_target_line < int(lines[current_line][2:5]):
            back_lines.append(j)
            undeclared_lines =
backward_branch_execute(new_vars,undeclared_lines,inner_target_line,j,back_lines)
    else:
        back_lines.append(j)
        undeclared_lines =
backward_branch_execute(new_vars,undeclared_lines,inner_target_line,j,back_lines)
```

```
# in backward_branch_execute function
inner_target_line = int(line_j[2:5])
new_vars = deep_copy_dict(vars)
```

```
if inner_target_line > j:
    back_lines.append(current_line)
    undeclared_lines =
forward_branch_execute(new_vars,undeclared_lines,inner_target_line,current_line,ba
ck_lines)
elif inner_target_line < j:
    if j < current_line and inner_target_line < target_line:
        back_lines.append(j)
        undeclared_lines =
backward_branch_execute(new_vars,undeclared_lines,inner_target_line,j,back_lines)
```

The input is using sys.stdin, which is standard input, you should input the code through terminal. And the output just uses print, which can be seen in the terminal